Objektno programiranje

Tokrat še bolj kot za ostale kratke zapiske velja, da predpostavljam, da bralec snov že pozna iz drugih jezikov. Torej, da mu objektno programiranje ni tuje in mu moramo povedati le, kako se te stvari zapiše v Pythonu.

Definirali bomo razred `Turtle`, z metodami

- `forward(a)`, `back(a)`, ki jo premakneta za `x` naprej oz. nazaj;
- `turn(a)`, ki jo obrne za `a` stopinj levo;
- `left()`, `right()`, ki jo obrneta za 90 stopinj levo oz. desno.

Iz tega razreda bomo izpeljali razred `TurtleProfessional`. Ta nam omogoča, da tiste stvari, ki so v običajni želvi sicer zastonj, plačujemo. Poleg naštetih ima še metodi:

- `set_cost(a)`, ki nastavi ceno za prehojeno enoto;
- `get_total()`, ki pove, koliko smo želvi dolžni.

Želva je v začetku na koordinatah (0, 0) in gleda desno. Privzeta cena plačljive želve je 1 na enoto premika.

```
from math import sin, cos, radians

class Turtle:
    def __init__(self):
        self.x = self.y = 0
        self.heading = 0

    def forward(self, a):
        self.x += a * cos(self.heading)
        self.y += a * sin(self.heading)

    def backward(self, a):
        self.forward(-a)

    def turn(self, a):
        self.heading += radians(a)

    def left(self):
        self.turn(90)

    def right(self):
        self.turn(-90)


class TurtleProfessional(Turtle):
    def __init__(self):
        super().__init__()
```

```
        self.cost = 1
        self.total = 0

    def forward(self, a):
        super().forward(a)
        self.total += abs(a) * self.cost

    def set_cost(self, a):
        self.cost = a

    def get_total(self):
        return self.total
```

Preden povemo kaj več, se samo še zapeljimo z eno želvo.

```
ana = TurtleProfessional()
ana.forward(10)
ana.left()
ana.set_cost(3)
ana.backward(5)
```

```
ana.get_total()
```

```
25
```

```
ana.x
```

```
10.0
```

```
ana.y
```

```
-5.0
```

Sintaksa

Očitno in pričakovana. Rečemo `class`, ime, dvopičje in naštejemo funkcije/metode. Tako kot nikjer drugje tudi tu ni potrebno deklarirati spremenljivk.

Če je razred izpeljan iz nekega drugega razreda, navedemo ta razred v oklepaju. Občutek imam, da imajo to jeziki v zadnjem času rajše, v Pythonu pa ima dvopičje itak še čisto drug pomen.

Poljem razreda (`x`, `y`, `heading` ...) se v Pythonu reče atributi (*attribute*). Potrebe po slovenjenju ne vidim, saj je beseda v SSKJ brez klicajev. Poleg bi lahko prevod "lastnost" zamenjevali s *property*, ki ima nekoliko drugačen pomen.

Večkratno dedovanje

O tem se kar takoj pogovorimo. En razred ima lahko tudi več prednikov; navedemo jih v oklepaju in ločimo z vejicami.

Večkratno dedovanje je običajno zelo slaba ideja, zato ga veliko jezikov sploh ne podpira. To je ena od tistih stvari, ki jih je celo Java naredila prav; katere so ostale tri, sem pozabil. Python to dopušča, vendar to izjemno rekdo vidimo; spomnim se samo enega razreda, za katerega vem, da je izpeljan iz dveh, vendar gre za *wrapper* okrog C-jevske knjižnice.

Zakaj Python potem to ima, če celo Java ve, da to ni OK in če to vejo tudi programerji?

Zelo redko (po mojih izkušnjah: nikoli) se ne zgodi, da bi en razred hkrati spadal v dve vrsti razredov. Ali si `Oseba` ali pa si `Kvadrat` in nihče, noben izpeljan razred, ne more biti oboje.

Pač pa je večkratno dedovanje zelo uporabno in tudi neškodljivo, če ga ne dojemamo, kot da nekdo spada v dve vrsti reči, pač pa z dodatnim razredom dodamo ali obljubimo neko funkcionalnost.

En vzorec je mixin, s katerim razredu dodamo neko funkcionalnost. V bistvu gre za razred, ki definira neke dodatne metode, morda tudi kakšne z njimi povezane atribute, ne gre pa za nek funkcionalen, uporaben razred. Tipično (ne pa nujno) mixini tudi nimajo kakšne posebne hierarhije. V projektu, na katerem delam, imamo mixin za multi threading. Če nek razred "izpelješ" *tudi* iz tega mixina, dobi dodatne metode za delo s threadi. Imamo mixin za progress bar, ki grafičnim elementom doda progress bar. Mixin, ki kontrolira določene atribute...

Drugi vzorec, ki ga včasih uporabljamo v Pythonu, je označevanje razredov. Razred izpeljemo iz nekega razreda, če želimo povedati, da zna določene stvari. Razrede včasih izpeljemo *tudi* iz razredov v `typing` ali `collections.abc` (abc = abstract base class). Razred izpeljan iz `collections.abs.Collection`, recimo, o sebi trdi, da je čezenj možno z zanko `for`, da zna povedati svojo dolžino in ima operator `in`. (Več v dokumentaciji.)

Oba vzorca spominjata na *interface* iz Jave in C#. Javin interface, če ga kot nekdo, ki nikoli ni veliko programiral v Javi, v zadnjih desetih letih pa sploh nič, bolj spominja na abstraktne razrede. Z njim *obljubimo* (oz. zahtevamo), da bo razred imel določene metode. Prevajalnik to preverja. Pythonova deklaracija abstraktnega podatkovnega tipa ima podoben učinek, vendar tega nihče ne preverja.

C# pa od različice 8.0 (iz leta 2019) omogoča, da v interface določimo privzete implementacije metod. To so v bistvu mixini.

Konstruktor in destruktor

Konstruktor je metoda z imenom `__init__`. To je samo ena izmed skoraj 50 posebnih metod, ki se začnejo in končajo z dvema podčrtajema. (Ne zmotite se in ne napišite samo enega.)

Konstruktor seveda ni obvezen. Napišemo ga, če imamo v njem kaj povedati.

Konstruktor ima lahko tudi dodatne argumente. Te pač navedemo, ko naredimo objekt. Če bi konstruktor razreda `Turtle` definirali z

```
def __init__(self, name):
    ...
```

bi ga klicali z

```
ana = Turtle("Ana")
```

Destruktorjev ne pišemo skoraj nikoli. In v bistvu niso zares v paru z `__init__`, temveč z drugim konstruktorjem.

Python ima v bistvu dva konstruktorja. `__init__` kot argument prejme `self`, saj je objekt že narejen. Naloga `__init__`-a je, da inicializira objekt, pripravi njegove atribute. Metoda, ki v resnici naredi objekt, torej ustvari, vrne `self` pa je `__new__`. Ta metoda kot argument prejme razred in (praviloma) vrne objekt tega razreda.

Konstruktor, `__del__` se pokliče, preden objekt odide iz pomnilnika. Kdaj se bo to zgodilo, pa ne vemo. Takrat, ko ga bodo smetarji naložili in odpeljali. Pri nas običajno okrog osme ure zjutraj; v ponedeljek organske, v torek embalažo in tako naprej. Večina objektov se razblini, ki nanje ne kaže nobeno ime več; če so udeleženi v kakšni skupinski dejavnosti, ki obsega cikle, pa lahko tudi kdaj kasneje.

Ne le, da ne vemo, kdaj se bo `__del__` poklical, temveč v njem navadno tudi nimamo česa početi. Redki programerji v Pythonu so že kdaj napisali konstruktor. Pretiravam, a ne veliko. :) Zase lahko povem, da ga že precej let nisem napisal.

Argument `self`

V jeziku (jezikih), iz katerega poznate objektno programiranje, ste gotovo navajeni spremenljivke z imenom `this` ali `self`. V Pythonu se ji reče `self`. Python se od jezika, ki ga poznate, najbrž razlikuje po tem, da je ta `self` vedno ekspliciten.

1. `self` je vedno prvi argument metode. V C++, C#, Java, Javascript ... se `this` preprosto pojavi. Funkcija ga dobi kot enega izmed argumentov, programer pa tega niti ne ve. Tu ve in ga mora navesti. Razen, kadar ga ne, ker gre za statično ali razredno metodo. A o tem bomo kasneje (če sploh).

2. Ob dostopanju do atributov in ob klicih metod moramo vedno navajati `self`. Ttako kot v Javascriptu (kjer je sicer `this`) in drugače kot v C++ in iz njega izpeljanih jezikih (C++ sam pa je to pobral od Simule 67), kjer `this` napišemo, kadar se nam zdi.

   Logika je v tem: ko smo šofirali Ano, smo pisali `ana.forward`. Vedno smo povedali *kdo* naj gre naprej. Zakaj bi znotraj metod delali izjemo?

Praktičen razlog za to, je, da so v C++ in podobnih jezikih atributi definirani. Tam se ve, da obstaja atribut `x`, torej se bo ime `x` znotraj metod nanašalo na `this->x`. V Pythonu pa že v konstruktorju ne bi bilo jasno, ali hočemo z `x = 0` nastaviti lokalno spremenljivko ali atribut.

Poleg tega pa ima ta odločitev kup lepih posledic, ki pa jih lahko razumemo samo, če vemo, kako je Python narejen - to je, kako pravzaprav poišče objekt, na katerega se nanaša ime, kako v hierarhiji razredov poišče metodo ipd.

Tale `self` boste malo pozabljali, potem pa se boste navadili. Če sam zelo dolgo programiram samo v kakem drugem jeziku, se včasih še sam spozabim.

Klicanje podedovanih metod

`super()` vrne objekt, ki je takšen kot `self`, vendar se vede, kot da bi bil objekt razreda, iz katerega je izpeljan ta razred. Kaj v resnici vrne `super()` in kako deluje, je magija, v katero se, priznam, tudi sam nisem najglobje spustil. Vedno mi je zadoščalo vedeti, da podedovani `forward` pokličem s `super().forward` in podedovani konstruktor s `super().__init__`.

Privatne metode

Python si je izmislil Nizozemec. Če ste se kdaj sprehajali po Amsterdamu, veste, kako (oz. po čem) tam diši. Skoraj kot v San Franciscu. Nizozemci ne skrivajo.

Python nima privatnih metod. Vse je javno. Pač pa se severni narodi bolj držijo družbenih norm.

- Če ime metode ali atributa začnemo s podčrtajem, to pomeni, da gre za privatno zadevo. Še vedno jo vsi vidijo in uporabljajo, vendar na lastno odgovornost. Če gre za kakšno knjižnico in kličeš metodo, katere ime se začne s podrčtajem (enako velja tudi za funkcije znotraj modulov), ti nihče ne garantira, da bo ta metoda v prihodnji različici še obstajala, sprejemala enake argumente in počela isto kot prej. In obratno: ko pišemo svoje razrede, damo metodam, ki jih kanimo obdržati in deliti s svetom, normalna imena, takšnim, v zvezi s čemer na garantiramo ničesar, pa damo na začetek imen podčrtaj.

```
class A:
    def _ne_klici_me(self):
        print("As mogu?")

a = A()
a._ne_klici_me()
```

```
As mogu?
```

- Če damo na začetek metode dva podčrtaja, pa se bo celo Python malo potrudil, da oteži klice.

```
class A:
    def __res_me_ne_klici(self):
        print("A ti je ratal?")

a = A()
a.__res_me_ne_klici()
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-7-4b5b4035e01e> in <module>
      4
      5 a = A()
----> 6 a.__res_me_ne_klici()

AttributeError: 'A' object has no attribute '__res_me_ne_klici'
```

```
a._A__res_me_ne_klici()
```

```
A ti je ratal?
```

Dostop do atributov

Tako kot Python ne skriva metod, tudi ne skriva atributov. Zgoraj smo videli, da lahko dostopamo do `ana.x` in `ana.y`. Pisanje metod, s katerimi preberemo ali nastavimo vrednost atributa je stvar okusa. Nekatere knjižnice so tozadevno zelo striktne, druge tega nimajo.

Na tem mestu vas bo morda zanimalo, ali obstajajo propertyji, getterji in setterji. Da, ampak dajmo o tem kasneje.

Atribute smo dodajali v konstruktorju. *Tako se spodobi.* Lahko pa jih dodajamo tudi v drugih metodah. Poljubna metoda lahko reče `self.nekaj = 42`. *Tako se ne spodobi.* V tem primeru atribut obstaja samo, če pokličemo tisto metodo, sicer pa ne. To nekaj časa deluje, potem pa nekaj spremenimo, malo drugače pokličemo metode, in neha. `pylint` vas bo zato opozoril, če v neki metodi nastavljate atribut, ki ne dobi neke vrednosti že v konstruktorju.

Atribute lahko nastavljamo celo od zunaj.

```
ana.name = "Ana"
```

```
ana.name
```

```
'Ana'
```

Tega ni možno početi, zato, ker so to dodali v jezik in bi bilo to posebej dobra praksa, temveč zato, ker tega niso vzeli iz jezika, preprečili. Točneje, ker bi bilo glede na to, kako je Python narejen, to pravzaprav zelo težko preprečiti.

Virtualne metode

Vse metode so virtualne.

Profesionalna želva je definirala svoj `forward`. Ko `backward` (ki je definiran v razredu `Turtle` pokliče `forward`, se pokliče `forward`, ki je v razredu `TurtleProfessional`.

Ugibam, da v prevajanih jezikih ni tako, ker lahko prevajalnik pri ne-virtualnih metodah optimizira kodo, poleg tega pa virtualne metode zahtevajo dodaten vpogled v tabelo, v kateri so zbrani kazalci na virtualne metode, medtem ko se za ne-virtualne že v času prevajanja (točneje: povezovanja, *link*) ve, kje v programu se bodo nahajale.

V jezikih, ki niso prevajani, ali pa vsaj v Pythonu te razlike ni.

Zaprti razredi

V nekaterih jezikih je razred možno označiti kot `final` - iz takega razreda ni možno izpeljevati novih razredov. To je dobra ideja.

Nekateri jeziki gredo še dlje: v Kotlinu je možno iz razreda izpeljevati nove razrede samo, če so označeni kot `open`. To je še boljša ideja.

V Pythonu ni ničesar od tega.

Python ima od različice 3.8 dekorator `@final`, s katerim označimo razred, iz akterega naj bi se ne dalo izpeljevati novih razredov, ali metodo, ki naj bi je ne bilo možno povoziti v izpeljanih razredih. Kot vse te stvari pa je tudi to le stvar dogovora. Gre za anotacijo tipa, tako kot lahko v Pythonu z `i: int` prisežemo, da bo `i int` in v naslednji vrstici iz gole hudobije v `i` shranimo niz. *I nikome ništa.* Pač pa na te dekoratorje prežijo razni linti in nas opozarjajo.

V projektu, na katerem delam, smo nekoč ugotovili, da bi radi zaprli vse razrede v določeni hierarhiji. Zato sem sprogramiral neko čarovnijo, s katero lahko naredimo tole:

```
class OurBaseClass:
    # naš razred, ki vsebuje čarovnijo, ki preprečuje, da bi iz razredov,
    # izpeljanih iz tega razreda, izpeljevali nove razrede
    ...

class A(OurBaseClass):
    ...

class B(A):  # tu Python javi napako, da razred A ni odprt
    ...

class C(OurBaseClass, openclass=True):
    ...
```

```
class D(C):  # to pa je dovoljeno, ker je C odprt
    ...
```

Je pa ta čarovnija prezapletena, da bi jo tule razlagal. Temelji na uporabi meta razredov. Na kratko, `OurBaseClass` ni tip tipa tipa, temveč tip nekega našega tipa.

Lastnosti (property)

Property je kot atribut, le da se v resnici vsakič znova izračuna.

```
class A:
    def __init__(self, k):
        self.k = k

    @property
    def dva_k(self):
        return 2 * self.k

a = A(5)
a.dva_k
```

```
10
```

```
a.k = 42
a.dva_k
```

```
84
```

Kako naredimo `property`, vidimo: okrasimo ga z dekoratorjem. (Kot večina (vsi?) drugi dekoratorji je tudi ta takšen, da bi ga, če še ne bi obstajal, lahko napisali sami. Samo malo več bi morali vedeti o delovanju razredov.)

Lastnosti pogosto uporabljamo, kadar želimo nek podatek skriti in predvsem, kadar želimo nadzirati njegovo nastavljanje.

Sestavimo razred, ki bo vseboval atribut `tabela`, katere dolžina bo vedno tolikšna, kolikor je vrednost njegovega atributa `dim`.

```
class Tabela:
    def __init__(self, dim):
        self._dim = dim
        self.tabela = [0] * dim

    @property
    def dim(self):
        return self._dim

    @dim.setter
```

```
    def dim(self, new_dim):
        if self._dim < new_dim:
            self.tabela += [0] * (new_dim - self._dim)
        else:
            self.tabela = self.tabela[:new_dim]
        self._dim = new_dim

t = Tabela(4)
t.tabela
```

```
[0, 0, 0, 0]
```

```
t.dim
```

```
4
```

```
t.dim = 10
t.dim
```

```
10
```

```
t.tabela
```

```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

Tako kot branje atributa `dim` v resnici pokliče funkcijo `dim`, ki smo jo okrasili s `property`, nastavljanje tega atributa pokliče funkcijo, ki smo jo okrasili z `@dim.setter` (in jo moramo napisati za prvo funkcjo, saj prvi dekorator ustvari drugega).

Tabela se je takrat, ko smo spreminili `dim`, sama od sebe podaljšala.

Lastnosti (*property*) lahko naredimo tudi drugače. Vsaj na dva načina. Mehanizacija, ki se skriva za tem, je kar zapletena; za vtis, si lahko ogledate dokumentacijo.

Atributi razredov

C++ in C# imata statične atribute (*static member*). Zakaj se jim reče statični, ne vem. V Pythonu so to atributi razredov.

```
class A:
    x = 12

    def f(self):
        return 42

a = A()
b = A()
```

Tako kot imata oba objekta metodo `f`, imata tudi atribut `x`.

```
a.f()
```

```
42
```

```
a.x
```

```
12
```

Če objekt definira svoj `x`, ima ta prednost pred atributom razreda.

```
a.x = 20
```

```
a.x
```

```
20
```

```
b.x
```

```
12
```

Skupni `x` je pač `A.x`.

```
A.x
```

```
12
```

```
A.x = 13
```

```
b.x
```

```
13
```

Tudi tu še vedno velja vse, kar smo risali s puščicami.

```
class A:
    t = [1, 2, 3]

a = A()
b = A()

a.t.append(4)
b.t
```

```
[1, 2, 3, 4]
```

```
a.t is b.t is A.t
```

```
True
```

```
a.t = [10, 11, 12]
b.t
```

```
[1, 2, 3, 4]
```

Tu je `a` dobil svoj `t`, `b.t` pa je še vedno, kar je bil prej.

Nekaj o tem, kako je to narejeno

Kje so shranjeni atributi

Razredi imajo slovar, `__dict__`, ki vsebuje metode (in atribute razreda). Objekti imajo slovar `__dict__`, ki vsebuje atribute (in tudi metode, ampak o tem bomo govorili posebej).

```
class A:
    def __init__(self, ime):
        self.ime = ime
        self.t = 13

    def f(self, x):
        print(f"f razreda A, objekta {self.ime}, x={x}")

    def g(self):
        print(f"g razreda A, objekta {self.ime}")

class B(A):
    def f(self, x):
        print(f"f razreda B, objekta {self.ime}, x={x}")

ana = B("Ana")
berta = B("Berta")
```

`ana` ima dva atributa, shranjena sta v slovarju `ana.__dict__`. `berta` prav tako in sta pač v `b.__dict__`.

```
ana.__dict__
```

```
{'ime': 'Ana', 't': 13}
```

Ko torej hočemo dobiti vrednost `ana.nekaj`, gre Python iskat to vrednost v `ana.__dict__`.

```
ana.__dict__["prijatelj"] = "Benjamin"
```

```
ana.prijatelj
```

```
'Benjamin'
```

Zapis `ana.prijatelj` je samo okrajšava za `ana.__dict__["prijatelj"]`. No, skoraj. Ker imamo tu še razrede.

Kako pridemo do metod

Metode so, očitno, shranjene v slovarju razreda.

```
A.__dict__
```

```
mappingproxy({'__module__': '__main__',
              '__init__': <function __main__.A.__init__(self, ime)>,
              'f': <function __main__.A.f(self, x)>,
              'g': <function __main__.A.g(self)>,
              '__dict__': <attribute '__dict__' of 'A' objects>,
              '__weakref__': <attribute '__weakref__' of 'A' objects>,
              '__doc__': None})
```

```
B.__dict__
```

```
mappingproxy({'__module__': '__main__',
              'f': <function __main__.B.f(self, x)>,
              '__doc__': None})
```

Tole niso čisto pravi slovarji, vendar se vedejo ... precej kot da bi bili. Ne moremo jim direktno spreminjati vrednosti (kot smo zgoraj dodali `"z"` v `b`-jev slovar), ampak tega niti ne potrebujemo.

Kaj se torej zgodi, ko napišemo `b.k` ali `b.f`? Tole

- Python pogleda, ali takšen ključ (`"k"`, `"f"`) obstaja v slovarju samega objekta (`b`-ja). Če, potem ga vrne.
- Sicer pogleda, ali to obstaja v slovarju razreda, ki mu pripada objekt. Če, je to to.
- Sicer pogleda v razred, iz katerega je ta razred izpeljan. In tako naprej.

Za vsak razred lahko izvemo, iz katerega razreda (oz. razredov) je izpeljan.

```
B.__bases__
```

```
(__main__.A,)
```

Še bolj zanimivo je, da lahko za vsak razred izvemo celo hierarhijo razredov nad njim. Da bo bolj zanimivo, dodajmo še en razred.

```
class C(B):
    ...
```

```
C.mro()
```

```
[__main__.C, __main__.B, __main__.A, object]
```

Vsi razredi imajo metodo `mro`, ki vrne seznam razredov, v katerih bo Python iskal metode (in atribute). Za razred `C` so to razredi `C`, `B`, `A` in na koncu `object`, iz katerih so izpeljani vsi razredi. (Metoda `mro`, recimo, je metoda razreda `object`.)

Sestavimo primerek razreda `C`.

```
c = C(3.14)
```

Ko torej napišemo `c.f` se zgodi naslednje.

```
def poisci(objekt, ime):
    if ime in objekt.__dict__:
        print(f"Našel v slovarju objekta; {ime}={objekt.__dict__[ime]}")
        return objekt.__dict__[ime]
    for cls in type(objekt).mro():
        if ime in cls.__dict__:
            print(f"Našel med metodami razreda '{cls.__name__}'; {ime}={cls.__dict__[ime]}")
            return cls.__dict__[ime]
    print(f"Ime '{ime}'' ne obstaja.")
    return None
```

```
poisci(c, "k")
```

```
Ime 'k'' ne obstaja.
```

```
poisci(c, "f")
```

```
Našel med metodami razreda 'B'; f=<function B.f at 0x7fc427951050>
```

```
<function __main__.B.f(self, x)>
```

```
poisci(c, "g")
```

```
Našel med metodami razreda 'A'; g=<function A.g at 0x7fc427946ef0>
```

```
<function __main__.A.g(self)>
```

```
poisci(c, "tega_ni")
```

```
Ime 'tega_ni'' ne obstaja.
```

Tule vidimo tudi, zakaj je tako dobro, da moramo vedno eksplicitno navesti `self`. Na ta način se bo `ana.x` sproži natančno isti mehanizem kot ob `self.x`. Edino, v čemer se Pythonove metode razlikujejo od običajnih funkcij, je, da ob klicu `ana.f` `ana` postane prvi argument `f`-a.

Vezane metode

Funkcija v Pythonu je preprosto objekt, ki se ga da poklicati.

```
def f(x):
    return x * 2
```

```
f
```

```
<function __main__.f(x)>
```

Podobno so tudi metode samo objekti, ki se jih da poklicati. Le, da imajo nek pripet `self`.

```
ana.f
```

```
<bound method B.f of <__main__.B object at 0x7fc42792c890>>
```

`ana.f` ni `function` temveč `bound method`. Vendar se vede kot ... pač funkcija. Lahko jo podamo funkcijam, ki pričakujejo funkcije, lahko jo priredimo spremenljivki, lahko jo, seveda, tudi pokličemo.

```
anin_f = ana.f
bertin_f = berta.f
```

```
anin_f(-5)
```

```
f objekta razreda B, objekta Ana, x=-5
```

```
bertin_f(-5)
```

```
f objekta razreda B, objekta Berta, x=-5
```

Metoda `bertin_f`, ki je seveda isto kot `berta.f`, je torej že vezana na Berto. Podobno velja tudi za metode vdelanih objektov.

```
s = [1, 2, 3]
```

```
s.append
```

```
<function list.append(object, /)>
```

```
t = s.append
```

```
t(4)
```

```
s
```

```
[1, 2, 3, 4]
```

Nevezane metode

Če je metoda lahko vezana, je očitno lahko tudi nevezana. To so metode razredov - ne objektov.

```
B.f
```

```
<function __main__.B.f(self, x)>
```

Nevezana metoda je dejansko funkcija. Pač funkcija, ki jo je potrebno poklicati z - v tem primeru - dvema argumentoma.

```
B.f(ana, 13)
```

```
f objekta razreda B, objekta Ana, x=13
```

Če hočemo, jo lahko pokličemo tudi s kakim drugim objektom in Python se niti ne bo pritožil, razen če se bo moral.

```
B.f([1, 2, 3], 5)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-52-f3adb971e019> in <module>
```

```
----> 1 B.f([1, 2, 3], 5)

<ipython-input-27-9de45a5a21d5> in f(self, x)
     12 class B(A):
     13     def f(self, x):
---> 14         print(f"f objekta razreda B, objekta {self.ime}, x={x}")
     15
     16 ana = B("Ana")

AttributeError: 'list' object has no attribute 'ime'
```

Tule ga je zmotilo, da `self` (torej: seznam `[1, 2, 3]` nima atributa ime.

Lahko pa mu podtaknemo kakršnokoli reč, ki takšen atribut ima.

```
class F:
    def __init__(self):
        self.ime = "Anon"

d = F()

B.f(d, 13)
```

```
f objekta razreda B, objekta Anon, x=13
```

Objekt `d` nima popolnoma nobene zveze z razredom `B`, vendar ga lahko podamo kot argument `B`-jevi metodi `f`.

To je seveda grdo in tega ne počnemo.

- Zakaj se da? Zato ker nihče ne preverja. :)
- Zakaj to kažem? Zato da boste razumeli, kako deluje. Ne zato, da bi uporabljali.

Kje pa je to uporabno. Oh, je, seveda je. Recimo. Imejmo niz

```
s = "1 2 3"
```

Če ga hočemo razkosati, bomo navadno poklicali

```
s.split()
```

```
['1', '2', '3']
```

Je pa to isto kot

```
str.split(s)
```

```
['1', '2', '3']
```

Enkrat imamo pač vezano metodo

```
s.split
```

```
<function str.split(sep=None, maxsplit=-1)>
```

ki jo pokličemo brez argumentov (`s.split()`), drugič nevezano metodo

```
str.split
```

```
<method 'split' of 'str' objects>
```

ki jo pokličemo tako, da ji podamo niz, ki bo služil kot `self`, torej `str.split(s)`.

Drugi klic je nenavaden in ga ne uporabljamo, pač pa nam lahko pride prav, kadar želimo funkcijo `str.split` podati kaki drugi funkciji, na primer funkciji `map`.

```
ss = ["1 2 3", "3 1", "3 5 8"]
list(map(str.split, ss))
```

```
[['1', '2', '3'], ['3', '1'], ['3', '5', '8']]
```

Monkey patching

Zdaj, ko vemo, kako vse skupaj deluje, bomo lahko razumeli delovanje "monkey patchinga". V nekaterih jezikih je to običajna reč (primer je Ruby, na nek način pa tudi starejši JavaScript). V Pythonu to počnemo izjemoma, pogosto pa to delamo v testih.

```
class A:
    def __init__(self, k):
        self.k = k

    def f(self, x):
        return x + self.k

a = A(3)

a.f(5)
```

```
8
```

Objekt `a` (oziroma razred `A`) očitno nima metode `g`. Lahko pa jo dobi.

```
def krat(self, x):
    return x * self.k

A.g = krat

a.g(5)
```

```
15
```

Kaj se je zgodilo? Nič posebnega: metoda `g` se je pojavila v `A.__dict__` in se vede, kot da je tam že od vedno.

```
A.__dict__
```

```
mappingproxy({'__module__': '__main__',
              '__init__': <function __main__.A.__init__(self, k)>,
              'f': <function __main__.A.f(self, x)>,
              '__dict__': <attribute '__dict__' of 'A' objects>,
              '__weakref__': <attribute '__weakref__' of 'A' objects>,
              '__doc__': None,
              'g': <function __main__.krat(self, x)>})
```

Seveda lahko tudi spremenimo katero od obstoječih metod.

```
def minus(self, x):
    return x - self.k

A.f = minus

a.f(5)
```

```
2
```

Kot rečeno, tega nikoli ne delamo v običajni kodi, pač pa to uporabljamo pri testiranju kode, kjer pri testiranju določenih metod zamenjamo nekatere druge metode z "mocki", ki sprejemajo oz. vračajo določene testne vrednosti, ali pa uporabljamo te mocke zato, da preverimo, ali je bila neka metoda res poklicana.

Več o tem pa, ko/če se bomo kdaj pogovarjali o testiranju.